

# Timed Games for Computing Worst-Case Execution-Times

Franck Cassez\*

National ICT Australia & CNRS  
The University of New South Wales  
Sydney, Australia

**Abstract.** In this paper we introduce a framework for computing upper bounds yet accurate WCET for hardware platforms with caches and pipelines. The methodology we propose consists of 3 steps: 1) given a program to analyse, compute an equivalent (WCET-wise) abstract program; 2) build a timed game by composing this abstract program with a network of timed automata modeling the architecture; and 3) compute the WCET as the optimal time to reach a winning state in this game. We demonstrate the applicability of our framework on standard benchmarks for an ARM9 processor with instruction and data caches, and compute the WCET with UPPAAL-TiGA. We also show that this framework can easily be extended to take into account dynamic changes in the speed of the processor during program execution.

## 1 Introduction

Embedded real-time systems are composed of a set of tasks (software) that run on a given architecture (hardware). These systems are subject to strict timing constraints and these constraints must be enforced by a scheduler. Designing an effective scheduler is possible only if some bounds are known about the execution times of each task. For simple scheduling algorithms e.g., non preemptive, the knowledge of the *worst-case execution-time* (WCET) is sufficient to design a scheduler. For more complex scheduling algorithms with preemption or shared resources, the WCET for each task might not give rise to the WCET for the entire system though. This is why most critical embedded systems rely on a rather simple scheduling algorithm. Performance wise, determining tight bounds for WCET is crucial as using rough over-estimates might either result in a set of tasks being wrongly declared non schedulable or a lot of computation time might be wasted in idling cycles and loss of energy/power.

**The WCET Problem.** The execution time,  $\text{time}(p, d, H)$ , of a program  $p$ , with input data  $d$  on the hardware  $H$ , is measured as the number of cycles of the fastest component of the hardware i.e., the processor. Data take their values

---

\* Author supported by a Marie Curie International Outgoing Fellowship within the 7th European Community Framework Programme.

in a finite domain  $\mathcal{D}$ . The program is given in binary code or equivalently in the assembly language of the target processor<sup>1</sup>. The *worst-case execution-time* of program  $p$  on hardware  $H$  is defined by:

$$\text{WCET}(p, H) = \sup_{d \in \mathcal{D}} \text{time}(p, d, H).$$

The WCET problem asks the following: Given  $p$  and  $H$ , compute  $\text{WCET}(p, H)$ .

In general, the WCET problem is undecidable because otherwise we could solve the halting problem<sup>2</sup>. However, for programs that always terminate and have a bounded number of paths, it is obviously (theoretically) computable. Indeed the possible runs of the program can be represented by a finite tree. Notice that this does not mean that the problem is tractable though.

If the input data are known or the program execution time is independent from the input data, the tree contains a single path and it is usually feasible to compute the WCET. Likewise, if we can determine some input data that produces the WCET (this might be as difficult as computing the WCET), we can compute the WCET on a single-path program.

It is not often the case that the input data are known or that we can determine an input that produces the WCET. Rather the (values of the) input data are unknown, and the number of paths to be explored might be extremely large: for instance, for a Bubble Sort program with 100 data to be sorted, the tree representing all the runs of the (assembly) program on all the possible input data has more than  $2^{50}$  nodes. Although symbolic methods (e.g., using BDDs) can be applied to analyse some programs with a huge number of states, they will fail to compute the exact WCET on Bubble Sort by exploring all the possible paths.

Another difficulty of the WCET problem stems from the more and more complex architectures embedded real-time systems are running on. They usually feature a multi-stage *pipeline* and a fast memory component like a *cache*, and they both influence in a complicated manner the WCET. It is then a challenging problem to determine a precise WCET even for relatively small programs running on complex architectures.

***Methods and Tools for the WCET Problem.*** The reader is referred to [1] for an exhaustive presentation of the WCET computation techniques and tools. There are two main classes of methods for computing WCET.

- Testing-based methods. These methods are based on experiments i.e., running the program on some data, using a simulator of the hardware or the real platform. The execution time of an experiment is measured and, on a large set of experiments, a maximal and minimal bound can be obtained. The

<sup>1</sup> When we refer to the “source” code, we assume the program  $p$  was generated by a compiler, and refer to the high-level program (e.g., in C) that was compiled into  $p$ .

<sup>2</sup> Note this is true even for input data ranging over a finite domain, and can be proved using König’s Lemma.

maximal bound computed this way is *unsafe* as not all the possible paths have been explored. These methods might not be suitable for safety critical embedded systems but they are versatile and rather easy to implement.

RapiTime [2] (based on pWCET [3]) and Mtime [4] are measurement tools that implement this technique.

- Verification-based methods. These methods often rely on the computation of an *abstract* graph, the control flow graph (CFG), and an abstract model of the hardware. Together with a static analysis tool they can be combined to compute WCET. The CFG should produce a super set of the set of all feasible paths. Thus the largest execution time on the abstract program is an upper bound of the WCET. Such methods produce *safe* WCET, but are difficult to implement. Moreover, the abstract program can be extremely large and beyond the scope of any analysis. In this case, a solution is to take an even more abstract program which results in drifting further away from the exact WCET.

Although difficult to implement, there are quite a lot of tools implementing this scheme: Bound-T [5], OTAWA [6], TuBound [7], Chronos [8], SWEET [9] and aiT [10,11] are static analysis-based tools for computing WCET.

The verification-based tools mentioned above rely on the construction of a control flow graph, and the determination of loop bounds. This can be achieved using user annotations (in the source code) or sometimes inferred automatically. The CFG is also annotated with some timing information about the cache misses/hits and pipeline stalls, and paths analysis is carried out on this model e.g., by Integer Linear Programming (ILP). The algorithms implemented in the tools use both the program and the hardware specification to compute the CFG fed to the ILP solver. The architecture of the tools themselves is thus monolithic: it is not easy to adapt an algorithm for a new processor. This is witnessed by *WCET'08 Challenge Report* [12] that highlights the difficulties encountered by the participants to adapt their tools for the new hardware in a reasonable amount of time.

***WCET and Model-Checking.*** Surprisingly enough, only a few tools use model-checking techniques to compute WCET. Considering that (i) modern architectures are composed of *concurrent* components (the stages of the pipeline, caches) and (ii) these components *synchronize* and synchronization depends on *timing constraints* (time to execute in one stage of the pipeline, time to fetch a data from the cache), formal models like *timed automata* [13] and state-of-the-art *real-time model-checkers* like UPPAAL[14,15] appear well-suited to address the WCET problem.

It has previously been claimed [16] that *model-checking* was not adequate to compute WCET, but this statement has since been revised. In [17], A. Metzner showed that model-checkers could well be used to compute safe WCET on the CFG for programs running on pipelined processors with an instruction cache.

In [18], B. Hubert and M. Schoeberl consider Java programs and compare ILP-based techniques with model-checking techniques using the model-checker

UPPAAL. Model-checking techniques seem slower but easily amenable to changes (in the hardware model). The recommendation is to use ILP tools for large programs and model-checking tools for code fragments.

More recently, the TASM toolset [19] (M. Ouimet & K. Lundqvist) has been used to compute WCET with UPPAAL: the TASM machine is a high level machine not featuring pipelining nor caches and computing the WCET amounts to finding the longest path (timewise) in a timed automaton that specifies a tasks.

Another use of timed automata (TA) and the model-checker UPPAAL for computing WCET on pipelined processors with caches is reported in [20]. The framework METAMOC described in [20] (A. E. Dalsgard *et al.*) consists in: 1) computing a flow graph (FG) from a binary program, 2) composing this FG with a (network of timed automata) model of the processor and the caches. Computing the WCET is then reduced to a safety (or dually a reachability) property  $\text{AG}(\text{Time} \leq k)$  (reads “on all paths, the variable Time, global time, is less than  $k$ ”) that can be checked with UPPAAL.

The previous framework is extremely elegant yet has some shortcomings. Out of the 15 programs<sup>3</sup> of the Mälardalen University benchmarks only 7 can be analysed with a concrete instruction and data cache (Table .6.1, page 84 in [20]). It is also surprising that some single-path programs could not be analysed with concrete caches. The tool chain relies on a value analysis tool which fails on 3 of the 15 programs. It requires a specialised version of UPPAAL (not available) to avoid a binary search for computing the WCET.

**Our Contribution.** In this paper we use *timed game automata* (TGA) and UPPAAL-TiGA [21] (UPPAAL for timed games) to compute WCET. We model the WCET problem as a two-player timed game. Intuitively Player 1 is the program, and Player 2 is in charge of deciding the outcome of the *comparison* instructions (e.g., `cmp`, `tst` which set the branching conditions) that depend on the input data. As the choice of the input data is not controllable by Player 1, we obtain a two-player game. The problem we solve on this game is an *optimal time reachability problem*:

“What is the optimal time for Player 1 to reach the end of the program ?”

What is similar to the previously mentioned approach [20] (A. E. Dalsgard *et al.*) is the timed automata models for the caches<sup>4</sup> and pipeline stages i.e., the model of the architecture, but we use a totally different model for the program. We propose a new and very compact encoding of the program and pipeline stages’ states which enables us to compute the WCET for 13 out of the previous 15 programs<sup>5</sup> (see Table 1, page 28). Moreover, compared to METAMOC that uses a computer with 32GB RAM, we can compute the results on a laptop

<sup>3</sup> The benchmarks contain 35 programs. In [20], only 14 programs can be analysed with a concrete instruction cache and 7 with a concrete instruction and data cache.

<sup>4</sup> Note that a similar model is reportedly due to A. P. Ravn in [18].

<sup>5</sup> Say why 2 fails ...

computer (2Ghz Dual Core, 2GB RAM) within a few seconds. Using timed games instead of timed automata is also a major difference: the on-the-fly algorithm [22] implemented in UPPAAL-TiGA is different from the one running in UPPAAL, and it can also compute the *optimal time* (in the presence of adversary) to reach a designated state. Thus we do not need to do a binary search or use a tailored version of UPPAAL to compute the results.

We also show that taking into account processor speed variations is easy in our framework. This can be important as it is possible to adjust the speed of the processor depending on the program to be run. For some programs, the saved power can be upto 22% (see Table 1).

The advantages of our approach are many-fold (METAMOC [20] shares 1–3):

1. it is very easy to implement as it consists of two separate and independent phases: 1) computation of a model of the program to be analysed; this only requires a (formal) semantics of the assembly language of the target processor<sup>6</sup>; 2) computation of the WCET with UPPAAL-TiGA and the models for the caches, pipelines which specify the timing features. A model of a cache (e.g., always miss or FIFO) can be substituted by changing the cache component only (no need to recompute the model obtained in phase 1).
2. the design of the models for pipeline stages and caches can be stressed by simulating some simple samples programs; this enables us to get more confidence in the model of the hardware as this is not hidden in the analysis algorithm; this is especially important for concurrent architectures like pipelined processors that can be hard to describe;
3. UPPAAL or UPPAAL-TiGA can be used to simulate the program on the architecture. It is thus a quick way of obtaining a simulator for a given hardware;
4. we do not require annotations. Instead, we run a simulation of the program with some given bounds on the number of branching or a maximal number of states. If too many branchings are encountered, the user is required to provide a constraint for the corresponding instruction in the program to remove some infeasible paths;
5. we solve an *optimal time reachability problem* on the program  $p$  of the form: “what is the optimal time to enforce *termination* of program  $p$  ?”. This at once 1) proves that  $p$  terminates on every input data, and 2) computes the WCET. This could not be achieved in METAMOC [20] as the UPPAAL model contains priorities and deadlock freedom cannot be checked on models with priorities: thus if the safety property  $\text{AG}(\text{Time} \leq k)$  is satisfied, it does not mean that no deadlocks occurred; the deadlocks could be due to a flaw in the design of the pipeline model but in any case, it does not give a safe bound for the WCET as deadlocks have not been excluded.
6. it is easy to add *power* related constraints in the model e.g., processor speed variations;

---

<sup>6</sup> In contrast, the verification-based tools would need a description of the hardware to compute the CFG.

7. we also show that not every program instruction is worth simulating and some *abstraction* on the effect of some instructions can be safely done. For example, in the Fibonacci program, the content of the variable with the result is irrelevant for the computation of the WCET. It does not influence any branching nodes. We show how to check that an abstract program is *equivalent* to a concrete one and exemplify this on some of the benchmarks from Mälardalen University.

**Outline of the Paper.** In Section 2, we briefly introduce the ARM9 architecture and the assumptions we make on the assembly programs to be analysed. Section 3 describes how to encode an assembly program with non-deterministic choices into a game. In Section 4 we give the timed automata models of the architecture we use to compute the WCET. Section 5 gives an overview of the tool chain we propose and the components (compiler) we have designed together with some comments on the case studies presented in Table 1.

## 2 Concrete and Abstract Programs

**Program, Registers, Memory.** A *program*  $p$  is a list of instructions  $p = i_1, i_2, \dots, i_k$  and  $i_1$  is the initial instruction. The control usually goes from instruction  $i_k$  to  $i_{k+1}$  except for branching instructions that give the next instruction  $i_j$  to be performed. Each instruction performs some basic operations (arithmetic, logic, memory load or store, branching) and has a duration which gives the amount of time it takes in each stage of the pipeline of the processor<sup>7</sup>. We assume the duration is independent from the content of the operands of the instructions<sup>8</sup>. In the sequel we use the variable  $\iota$  to denote an instruction of  $p$ .

The hardware on which  $p$  runs has a pool of *registers* (different from the main memory and the caches). We let  $\mathcal{R} = \{r_0, \dots, r_k\}$  be the set of registers. For example on the ARM9 [23] processor there are 16 registers. A designated register **pc** contains the program counter and points to the next instruction to be performed (register 15 on the ARM9).

We let  $\mathcal{M} = \{m_1, m_2, \dots, m_n\}$  be the set of memory cells' addresses used by the program (we assume the program can access  $\mathcal{M}$ ). The content of the memory cells and registers is in a finite domain  $\mathcal{D}$  (e.g., 32 bit integers).

**Semantics.** When program  $p$  runs on input data  $d$ , it generates a computation that changes the values of the registers and memory cells.

A state (of the computation of  $p$ ) is given by a mapping  $v : \mathcal{R} \cup \mathcal{M} \rightarrow \mathcal{D}$  and we let  $\mathcal{V}$  be the set of states.

<sup>7</sup> A particular case is a processor with one stage.

<sup>8</sup> This is not always the case as for instance the duration of the instruction **mull** (multiplication on long integers) on the ARM9 depends on how large one of the operand is. However, we can always take the longest duration to obtain a safe upper bound of the WCET.

Performing an instruction results in a state change, and is deterministic. Given an instruction  $\iota$  ( $\iota$  including the operands and can be thought of as the *code* of an assembly instruction), the semantics of  $\iota$  is a mapping  $\llbracket \iota \rrbracket : \mathcal{V} \rightarrow \mathcal{V}$ .

As the program counter is in one of the registers, the semantics of a program  $p$  is completely determined by the current state of the computation. From a state  $v$ , the next state (in the computation of  $p$ ) is  $v'$  and we denote this  $v \rightarrow v'$ .  $v'$  is given by  $\llbracket \iota \rrbracket(v)$  where  $\iota = i_{\mathbf{pc}}$  (we use  $\mathbf{pc}$  both for the register and the content of this register to avoid hefty notations).

For branching instructions, the control is determined by the *status bits* and we assume there are also part of the  $\mathbf{pc}$  register.

*Remark 1.* We assume  $\mathbf{pc}$  is incremented by 1 after each instruction (except for branching instruction). In an actual computer, it is incremented by the *word size* but these details are irrelevant at this stage.

**Side Effects of an Instruction.** Each instruction reads from and writes to some subset of registers. We let  $\text{regR}(\iota)$  (resp.  $\text{regW}(\iota)$ ) be the set of “read from” (resp. “written to”) registers for instruction  $\iota$ .

Each instruction can also read or write to main memory cells. We let  $\text{memR}(\iota)$  (resp.  $\text{memW}(\iota)$ ) be the set of memory cells addresses read from (resp. written to) by instruction  $\iota$ .

```
00000000 <main>:
0: e3a00009 mov     r0, #9 ; 0x9
4: eaffffff b       8 <binary_search>

00000008 <binary_search>:
8: e92d4030 stmdb   sp!, {r4, r5, lr}
c: e59f4040 ldr     r4, [pc, #64] ;
10: e3a0e000 mov     lr, #0 ; 0x0
14: e3a0c00e mov     ip, #14 ; 0xe
18: e3e05000 mvn     r5, #0 ; 0x0
1c: e08e300c add     r3, lr, ip
20: e1a020c3 mov     r2, r3, asr #1
24: e7943182 ldr     r3, [r4, r2, lsl #3]
28: e0841182 add     r1, r4, r2, lsl #3
2c: e1530000 cmp     r3, r0           / eq le /
30: 05915004 ldreq   r5, [r1, #4]
34: 024ec001 subeq   ip, lr, #1      ; 0x1
38: 0a000001 beq     44 <binary_search+0x3c>
3c: c242c001 subgt   ip, r2, #1      ; 0x1
40: d282e001 addle   lr, r2, #1      ; 0x1
44: e15e000c cmp     lr, ip           / le /
48: c1a00005 movgt   r0, r5
4c: dafffff2 ble     1c <binary_search+0x14>
50: e8bd8030 ldmia   sp!, {r4, r5, pc}
54: 00000158 andeq   r0, r0, r8, asr r1
```

**Listing 1.1.** Binary Search Program

encode the result of the comparison at line 24: whether  $r3$  is “lower or equal” than  $r0$  and whether  $r3$  is “equal” to  $r0$ . This is indicated by the two predicates *eq* and *le* between  $/ \dots /$ . The address of the memory cell referenced at line 24 is determined by the previous outcomes of the comparison instruction at line 2c.

An example of an assembly program is given in Listing 1.1. This program performs a binary search on an array of 14 elements. Line 24 loads register  $r3$  with a value of the array at address  $v(r4) + (v(r2) * 8)$ . As we do not know the values of the array, the value of  $r3$  is unknown after this instruction.  $r0$  contains the value we are looking for, and is also unknown<sup>9</sup>. As a consequence, the comparison of line 2c is undetermined as the value of  $r3$  is unknown. The outcome of the comparison is used later in conditional instructions (e.g., *ldreq*  $r5$ ,  $[r1, \#4]$  and *subgt*  $ip, r2, \#1$ ) and branching instructions *beq* 44. Two *status bits* are needed to

<sup>9</sup> In the actual program it is 9 but it does not change the execution tree of the program.

**Runs.** A *run* of program  $p$  from state  $v_0$  (initial value of the input data) is the (unique) sequence of instructions performed by  $p$  from  $v_0$ :

$$\rho(p, v_0) = \iota_1 \quad \cdots \quad \iota_k \quad \cdots \quad \iota_n$$

with  $\iota_1 = i_1$ . The length of the run  $\rho(p, v_0)$  is  $|\rho(p, v_0)| = n$ . We assume that every run terminates, and that moreover, given  $p$ , there exists a constant  $K_p$  s.t.  $\forall v \in \mathcal{V}, |\rho(p, v_0)| \leq K_p$ . Intuitively, this means that all loops are bounded, and it implies that there is no run which encounters twice the same state.

The state after the subsequence  $\iota_1 \cdots \iota_k$  is determined by the composition of the semantics function of each instruction. If  $v_j$  is the state after instruction  $\iota_j$  then  $v_{j+1} = \llbracket \iota_{j+1} \rrbracket(v_j)$ , and  $v_0$  is the initial state.

**Execution Time of a Run.** If each instruction was performed one after the other, the execution-time of a run would be the sum of the execution times of each instruction.

On pipelined architectures with caches, the execution-time solely depends on:

1. the subsequences of instructions: pipeline *stalls* can occur, for instance because one instruction (e.g., in the execute stage) reads a register written to by the instruction in the next stage (e.g., memory stage).
2. the time to read or write a memory cell: instructions that require memory transfers (load and store) might take different durations if a *cache* is used, depending on whether the memory cell is already in the cache or not.

We let  $H$  denote the architecture of the system.  $H$  refers to the pipeline structure and timing specifications, the cache initial state, size, replacement policy and timing specifications, and the timing specifications of the main memory. The *execution-time* of a run  $\rho$  is completely determined by:

- the architecture  $H$ ,
- the duration of each instruction of  $\rho$  in each stage of the pipeline,
- the registers read from and written to, and memory cells read from or written to by each instruction of  $\rho$ .

The *duration* of a run  $\rho$  on architecture  $H$  is denoted  $\text{time}_H(\rho)$ . This function might be rather complex but is yet well-defined.

To formalize the previous informal definition, assume the architecture  $H$  is fixed. Let  $\rho = \iota_1 \cdots \iota_n$  and  $\rho' = \iota'_1 \cdots \iota'_n$  be two runs of program  $p$ . We say that  $\rho$  and  $\rho'$  are (time-wise) *H-equivalent* and write  $\rho \approx_H \rho'$  if for each  $1 \leq k \leq n$ :

- the duration of  $\iota_k$  in each stage of the pipeline is the same as the duration of  $\iota'_k$ ;
- the registers used as operands and memory cells referenced are also the same:  $\phi(\iota_k) = \phi(\iota'_k)$  for  $\phi \in \{\text{regR}, \text{regW}, \text{memR}, \text{memW}\}$ .

**Fact 1** If  $\rho \approx_H \rho'$  then  $\text{time}_H(\rho) = \text{time}_H(\rho')$ .

The *worst-case execution-time* for program  $p$  on architecture  $H$  is given by:

$$\text{WCET}(p, H) = \max_{v_0 \in \mathcal{D}} \text{time}_H(\rho(p, v_0)).$$



**Timing Anomalies.** *Timing anomalies* [1] can occur because of the complex architecture of the hardware  $H$ . The term refers to counter-intuitive observations in the sense that larger *local* execution-times may not result in larger *global* execution-times. *Pre-fetching* instructions can lead to such observations on some processors. This can also be observed on complex pipeline architectures (e.g., *out-of-order* execution of instructions).

On architectures that do not exhibit *timing anomalies*, the function  $\text{time}_H$  is in some sense *monotonic*.

For instance an achitecture  $H_\mu$  with an “always miss” cache (or equivalently no cache) will produce a WCET which is always greater than on an architecture  $H$  with a cache of size more than 1. As we consider worst-case execution-time, a *random* replacement policy for a cache is equivalent to an “always miss” cache. Let  $H_r$  denote a cache with random replacement policy, and  $H$  a regular cache (LRU, FIFO, semi-random replacement policy). The following holds:

**Fact 2**  $\text{WCET}(p, H) \leq \text{WCET}(p, H_\mu) = \text{WCET}(p, H_r)$ .

This implies that an over-approximation of  $\text{WCET}(p, H)$  can always be obtained using an equivalent architecture  $H'$  with an “always miss” cache.

The same remark applies for the pipeline of architecture  $H$ . If  $H'$  is the same as  $H$  with larger durations for each instruction at each stage, then  $\text{WCET}(p, H) \leq \text{WCET}(p, H')$ . If a pipeline *stall* in  $H$  implies a pipeline stall in  $H'$  for every program and every input data, then  $\text{WCET}(p, H) \leq \text{WCET}(p, H')$ .

Another interesting case is when a *branch* instruction is executed. If it is not a loop, the program fragment has a diamond shape: both branches join at some future point in the computation. If the local worst-case execution time is obtained by taking one side of the branch instruction, we can safely ignore the other side as it does not contribute (more) to the global worst-case execution-time.

The framework of this paper does handle timing anomalies, but some abstractions defined below are not safe for architecture exhibiting timing anomalies.

**Abstractions.** In this section we introduce some simple abstractions that can be made on a program  $p$ . The aim of this abstraction is to reduce the space needed to encode the state of the computation. We exemplify the usefulness of these abstractions on some benchmarks programs from Mälardalen University.

Listing 1.2 (Fig.1) gives a C function computing the Fibonacci number  $n$ . Its assembly language version is given in listing 1.3. The control flow of the assembly version is controlled by lines 20, 24 and 30: register  $r2$  contains the loop variable  $i$  and is incremented at each round. Lines  $c$ , 10, 1c, 28 and 2c are not contributing to the program control flow. If we are only interested in the *execution-time* of this program, their *effects* can be safely abstracted away. We can replace them by equivalent instructions that modify only the **pc** register, with the same read/written registers (and memory cells if it happens to be a load/store instruction). For instance, instruction **mov** at line  $c$ , can be replaced by an *abstract* instruction **mov<sup>a</sup>** with:

```

1: int fib(int n)
2: {
3:     int i,Fnew,Fold,temp,ans;
4:     Fnew=1;Fold = 0;
5:     for(i=2;i<=30 && i<=n; i++)
6:     {
7:         temp=Fnew;
8:         Fnew=Fnew + Fold;
9:         Fold=temp;
10:    }
11:    ans=Fnew;
12:    return ans;
13: }

```

**Listing 1.2.** C Program

```

0: mov    r2, #2      ; 0x2
4: cmp    r2, r0
8: mov    ip, r0
c: mov    r0, #1      ; 0x1
10: mov    r1, #0      ; 0x0
14: movgt  pc, lr
18: add    r2, r2, #1   ; 0x1
1c: mov    r3, r0
20: cmp    r2, #30      ; 0x1e
24: cmple  r2, ip
28: add    r0, r0, r1
2c: mov    r1, r3
30: ble    18 <fib+0x18>
34: mov    pc, lr

```

**Listing 1.3.** Assembly Code

**Fig. 1.** Fibonacci Program.

- $\llbracket \text{mov}^a \rrbracket(v) = v'$  with  $v'(r) = v(r)$  for each register different from **pc** and  $v'(\text{pc}) = v(\text{pc}) + 1$ ;
- the duration of  $\text{mov}^a$  in each stage of the pipeline is the same as  $\text{mov}$ ;
- the registers read from/written to by  $\text{mov}^a$  at line c are the same as the ones read from/written to by instruction  $\text{mov}$  at line c.

In the end, we can abstract away the values of registers **r0**, **r1** and **r3** and assume they are always 0 as no abstract instruction will modify them. The WCET of the abstracted program will be exactly the same as the concrete one.

The goal of this abstraction is to reduce the space needed to encode a state of the computation. Instead of encoding 7 registers, only 4 are relevant for the computation of the WCET.

A valid abstract program must simulate the execution tree of the concrete program. To be equivalent WCET-wise to the concrete program, it should also preserve the addresses of the referenced memory cells to ensure that cache hits/misses are preserved.

To formalize the previous notions, we first define *critical* instructions. A *critical* instruction is an instruction that:

- (i) either sets some status bits; it can be a comparison or test (e.g., **cmp**, **tst**) or an arithmetic instruction with the “s” flag on the ARM9 (e.g., a subtraction **subs r2, r2, #1**);
- (ii) or an instruction that references a memory cell e.g., **ldr r0, [r2, r3 lsl #2]** (load register **r0** with the content of memory cell  $r2 + (r3 \times 4)$ ).

Next we define *abstract* instructions. As exemplified for the **mov** instruction at line c previously, given an instruction  $\iota$ , the *abstracted instruction*  $\iota^a$  is defined by:

- the semantics of  $\iota^a$  is  $\llbracket \iota^a \rrbracket(v) = v'$  with  $v'(x) = v(x)$  for each register  $x$  different from **pc** and each memory cell  $x$  in  $\mathcal{M}$ , and  $v'(\text{pc}) = v(\text{pc}) + 1$ ;

- the duration of  $\iota^a$  in each stage of the pipeline is the same as the duration of  $\iota$ ;
- the registers read from/written to by  $\iota^a$  are the same as the ones read from/written to by instruction  $\iota$ :  $\phi(\iota) = \phi(\iota^a)$  for  $\phi \in \{\text{reg}R, \text{reg}W, \text{mem}R, \text{mem}W\}$ .

Let  $p^a = i_1^a \cdots i_n^a$  be the abstract program that corresponds to  $p = i_1 \cdots i_n$ . An *abstraction mapping*  $\alpha$  is a mapping that associates with each (concrete) instruction  $\iota$  of  $p$ , either  $\iota$  (identity) or  $\iota^a$  ( $\alpha$  determines whether  $\iota$  is abstracted or not). We write  $\iota^a$  for  $\alpha(\iota)$ .

Let  $\rho(p, v_0) = \iota_1 \iota_2 \cdots \iota_k$  be a run of  $p$  from  $v_0$  and  $\rho(p^a, v_0) = \iota_1^a \iota_2^a \cdots \iota_k^a$  the corresponding  $\alpha$ -abstracted run. Let  $I_c(p, v_0) \subseteq \{1, 2, \dots, k\}$  be the set of indices s.t.  $j + 1 \in I_c(p, v_0) \iff \iota_{j+1}$  is a critical instruction in  $\rho(p, v_0)$ . Let  $v_j$  be the state after executing instruction  $j$  in  $\rho(p, v_0)$  and  $v_j^a$  be the state after executing abstract instruction  $j$  in  $\rho(p^a, v_0)$ .

The following Lemma states that, if the values of the registers read from/written to by any critical instruction (in  $\rho(p, v_0)$ ), are equal to the values of the same registers in the abstract execution, the execution time of the concrete and abstract run is the same.

**Lemma 1.** *If  $\forall j + 1 \in I_c(\rho(p, v_0))$ ,  $v_j(r) = v_j^a(r)$  for each  $r \in \text{reg}R(\iota_{j+1}) \cup \text{reg}W(\iota_{j+1})$  then  $\text{time}_H(\rho(p, v_0)) = \text{time}_H(\rho(p^a, v_0))$ .*

*Proof.* If the values of the operand registers of each critical instruction  $\iota_j$  are the same in the concrete and abstract runs before performing  $\iota_j$  and  $\iota_j^a$ , then:

1. the status bits that are set by the critical instruction have the same values in the concrete and abstract state;
2. the addresses of the memory cells referenced by the instruction are the same in the concrete and abstract run.

The concrete and abstract run are thus  $H$ -rquivalent, i.e.,  $\rho(p, v_0) \approx_H \rho(p^a, v_0)$ . By Fact 1, it follows that  $\text{time}_H(\rho(p, v_0)) = \text{time}_H(\rho(p^a, v_0))$ .  $\square$

If Lemma 1 holds for each run  $\rho(p, v_0)$  with  $v_0 \in \mathcal{D}$ , we say that  $p$  and  $p^a$  are  $H$ -equivalent and write  $p \approx_H p^a$ . In this case, by definition of the WCET, we have:

**Lemma 2.** *If  $p \approx_H p^a$  then  $\text{WCET}(p, H) = \text{WCET}(p^a, H)$ .*

**Context Independence.** As we cannot simulate  $p$  for every input data, we assume that the initial values of these data can be arbitrarily chosen. To formalize this, we use an extended domain for the values of the registers and memory cells:  $\mathcal{D} \cup \{\perp\}$  where  $\perp$  is a special *unknown* value. At the beginning of the computation, every register (except **pc**) and memory cell has its value set to  $\perp$ . The initial state is thus  $v_0$  with  $v_0(x) = \perp$  for  $x \in (\mathcal{R} \setminus \{\mathbf{pc}\}) \cup \mathcal{M}$  and  $v(\mathbf{pc}) = i_1$  where  $i_1$  is the address of the first instruction of program  $p$ .

We assume that for each program  $p$ , the addresses of the memory cells referenced during the course of the execution of the program, only depend on the

current state and are independent from the input data values. By this, we mean that the address referenced at each point in a run of a program is determined by some registers values that are known. These values may depend on the actual content of some memory cells because they influence the branching instructions, but once a branch is chosen, the addresses can be computed. An example is a binary search program: we have to determine whether a sorted array  $v$  contains a value  $s$ . The search continues as long as  $s$  has not been found.

The semantics of each instruction (next state) is extended to the extended domain  $\mathcal{D} \cup \{\perp\}$  as follows:

- for arithmetic and logical instructions, the value of the result of an instruction is  $\perp$  if the value of one of the operands is  $\perp$ ;
- for instructions that set the status bits, there might be more than one next state; if one operand is  $\perp$ , the next states are given by all the possible values of the status bits;
- for memory transfer instructions (load, store with addresses in  $\mathcal{M}$ ) the result in memory or register is always  $\perp$ . Nevertheless, for transfers involving the *stack* (a subset of the addresses in  $\mathcal{M}$ ), we keep track of the values pushed or popped. The stack is quite often used on call/return of a function, and abstracting the content of the stack would result in some infeasible paths, or even to references to forbidden memory cells.
- for branching instructions, there is one next state determined by the value of the target (unconditional branching) or by the status bits (conditional branching).

From the previous extended definitions, there might be more than one run from the initial extended state  $v_0$ . We denote  $p_\perp$  the non-deterministic program that corresponds to  $p$  on the extended domain. The semantics of  $p_\perp$  is a *tree*,  $\text{tree}(p_\perp)$  where the branches correspond to the choices of the status bits when required. Note that this tree might be unbounded.

An important property of this tree, is that if  $\rho(p, v_0)$  is a run of  $p$  on input data  $v_0$ , there is a path  $\rho'$  in  $\text{tree}(p_\perp)$  that satisfies  $\rho(p, v_0) \approx_H \rho'$ . Moreover, as we assume that the number of steps when running  $p$  is bounded by  $K_p$ , we can safely truncate the tree  $\text{tree}(p_\perp)$  and prune all nodes that are more than  $K_p$  steps apart from the root. Let  $\text{Runs}(p_\perp)$  denote the set of rooted paths in the tree  $\text{tree}(p_\perp)$ . We assume  $\text{tree}(p_\perp)$  has depth at most  $K_p$ . Let

$$\text{WCET}(p_\perp, H) = \max_{\rho \in \text{Runs}(p_\perp)} \text{time}_H(\rho).$$

As every run of  $p$  is simulated by a run  $p_\perp$ , we have:

$$\text{WCET}(p, H) \leq \text{WCET}(p_\perp, H).$$

Moreover, we can also define an abstract version,  $p_\perp^\alpha$ , of  $p_\perp$ , given an abstraction mapping  $\alpha$ . The definitions are extended to the extended domain. As before we have:

**Lemma 3.** *If  $p_\perp \approx_H p_\perp^\alpha$ , then  $\text{WCET}(p_\perp, H) = \text{WCET}(p_\perp^\alpha, H)$ .*

Combining Lemma 2 and Lemma 3, we have:

**Lemma 4.** *If  $p_{\perp} \approx_H p_{\perp}^{\alpha}$ ,  $\text{WCET}(p, H) \leq \text{WCET}(p_{\perp}^{\alpha}, H)$ .*

**Checking that  $p^{\alpha} \equiv_H p$ .** Checking whether  $p_{\perp} \approx_H p_{\perp}^{\alpha}$  can be done by building a *synchronized product* of  $p_{\perp}$  and  $p_{\perp}^{\alpha}$  and checking whether each state preceeding a critical instruction satisfies the condition of Lemma 1.

This is implemented in our framework (see Fig. 6) by generating a C++ file that performs this check.

Table 1, column *Abs* gives the ration of abstracted instructions for some programs (when we have chosen to abstract away some instructions). For some programs (`matmult` and `jfdcint`) the number of abstracted instructions is rather high. This indicates that the control flow is quite simple and governed by a small number of instructions.

Notice that this abstraction does not change the WCET of the program.

### 3 From Programs to Games

In this section we describe how to encode an assembly program into a game. The encoding can be applied to any assembly language but we give examples for the ARM9 processor.

Given a program  $p$ , we define a two-player game to model the runs of  $p_{\perp}$  defined in the previous section. Player 1 executes the instructions of  $p_{\perp}$ . The role of Player 2 is to set the values of the status bits when an instruction that modifies them is encountered and some operands have unknown values, the result is undetermined. The outcome is thus picked up non-deterministically.

On the ARM9 processor, there are 4 status bits. A simple encoding would be to have 4 boolean variables to model the value of each bit. As we let Player 2 choose the outcome, this corresponds to choosing four values for Player 2: N (negative), Z (zero), V (overflow) and C (carry). This could create  $2^4 = 16$  different next states and thus as many new potential branches in the game. Most of the time, it is not necessary to know the actual values of the 4 status bits. For instance the result of a comparison instruction `cmp r0, r1` with, say `r1` unknown, could be used later on only to check whether `r0 = r1`. In this case the value of the Z-status bit is required but the values of the other status bits are irrelevant.

To reduce the number of branches (choices of Player 2) in the game, we determine, for each instruction  $\iota$  that sets a status bit, the next instructions that depend on the result of  $\iota$ . This can be computed on the program  $p$ . For each instruction  $\iota$  that sets a status bits, we let  $flags(\iota)$  be the set of predicates used after  $\iota$ . For instance in the example code of Listing 1.3, Fig. 1 page 10, the result of the instruction `cmp r2, r0` line 4 is used at line 14, and the only predicate needed is `gt` (i.e., whether `r2 > r0`). In the worst case we still need 4 variables to encode the outcome of an instruction  $\iota$  that sets the status bits, but we reduce the choices of Player 2 to the predicates in  $flags(\iota)$ . In the previous examples, instead of having 16 branches, there will be only 2.

To model program  $p_{\perp}$  in UPPAAL we need:

- an array, `val`, of 16 variables for the registers of the ARM9 processor;
- 4 boolean variables for the status bits (we use `cmple`, `cmplt`, `cmpls`, `cmpeq` instead of the actual status bits N, Z, V and C, but this is equivalent);
- a *stack* of size  $K$  (the size of which has been determined in a previous stage).

Although the model-checker UPPAAL that we use is extremely efficient, we have to be careful when encoding  $p_{\perp}$ : some information can be encoded using variables, but they will be part of the *state* of the network of TA we build, and will be encoded in the BDD representation of each state. Some information are not dynamic but rather static (e.g., the *type* of an instruction  $\iota$ , or the registers read/written  $regR(\iota)$  and  $regW(\iota)$ ) and can be encoded using UPPAAL *functions*. This saves space as functions are not part of the encoding of a state. Given a program  $p_{\perp}$ , we define the functions:

- $SetStatusB : p \rightarrow \mathbb{B}$  which, given an instruction  $\iota \in p$ , returns TRUE if  $\iota$  sets some status bits (comparison instructions `cmp,tst` and instructions with the “s” flag like `subs`, `adds` etc);
- $cmpU : p \times \mathcal{V}_{\perp} \rightarrow \mathbb{B}$  which returns TRUE if the result of the instruction  $\iota$  in state  $v$  is unknown.

As a shorthand we write  $NDcmp(\iota, v) = SetStatusB(\iota) \wedge cmpU(\iota, v)$  and this indicates whether instruction  $\iota$ , when executed from state  $v$ , should be played by Player 2 (the status bits should be set but an operand is unknown).

In addition to this, we define another function  $update : \mathcal{V}_{\perp} \rightarrow \mathcal{V}_{\perp}$  which updates the values of the registers and the status bits if required: this function encodes the semantics of each instruction on the extended domain.

The result for the Fibonacci program of Listing 1.4 page 15 are given in Listings 1.5 and 1.6. These listings call for some comments:

- Listing 1.4 contains the assembly code generated by `objdump` after compiling the C program with `gcc`; the instructions that set status bits have been annotated (e.g., `lien 4 / le /`) by the predicates that should be set by the instruction (`le` in this case for instructions at lines 4, 20 and 24).
- Listing 1.5 contains the functions that determine whether the result of an instruction that sets the status bits is undetermined. `UNKNOWN` is a special value<sup>10</sup>. For instance, if the value of `r2` is unknown when executing instruction (hexadecimal) 20 (decimal 32), `cmpU` returns TRUE and `SetStatusB` as well.
- Listing 1.6 contains the updates of the registers in the extended domain. The updates of an instruction are performed only if it is not abstracted away (`is_abstracted` function, not given here, but we can assume for now it always returns FALSE.) The instruction `cmp r2,r0` (UPPAAL translation lines 13 to 20) sets the `cmple` variable according to the values of `r2` and `r0`. If at least one of the values of `r2` and `r0` is unknown, the value of `cmple` will

<sup>10</sup> We use an integer that is never used as an actual value in the content of any register.

be chosen right after the update step by Player 2, overriding the previous value.

The instruction `cmp r2,r0` is *unconditional*, and it has to be scheduled for execution. This is carried out by function `SET(-,-,-)` which sets 3 values (in the first stage of the pipeline, see section 4): the label of the instruction (4), the memory addresses referenced by the instruction (-1 indicates no memory addresses), and whether the instruction is scheduled or not (1 in this case).

For conditional instructions, e.g., `movgt pc, lr`, (UPPAAL translation lines 24 to 37), if the function `gt()` returns `TRUE`, the instruction is not scheduled (`SET(20,-1,0)`). Function `gt()` returns the complement value of `cmple` that has been set by the comparison instruction (or Player 2 if some operands were unknown) before.

The last parameter of `SET(-,-,-)` has no meaning for conditional branching instructions as they are always scheduled. We use it to indicate whether the condition evaluates to `TRUE` or `FALSE`. An example is instruction `ble 18` (UPPAAL translation lines 76 to 83 in listing 1.6). If the condition (function `le()`) evaluates to `TRUE` this parameter is `TRUE` and `FALSE` otherwise. This information is used to simulate pipeline *flushes* when a branch prediction is wrong.

```
00000000 <fib>:
0: e3a02002 mov r2, #2 ; 0x2
4: e1520000 cmp r2, r0 / le /
8: e1a0c000 mov ip, r0
c: e3a00001 mov r0, #1 ; 0x1
10: e3a01000 mov r1, #0 ; 0x0
14: c1a0f00e movgt pc, lr
18: e2822001 add r2, r2, #1 ; 0x1
1c: e1a03000 mov r3, r0
20: e352001e cmp r2, #30 ; 0x1e / le /
24: d152000c cmple r2, ip / le /
28: e0800001 add r0, r0, r1
2c: e1a01003 mov r1, r3
30: dafffff8 ble 18 <fib+0x18>
34: e1a0f00e mov pc, lr

00000038 <main>:
38: e1a0c00d mov ip, sp
3c: e92dd810 stmdb sp!, {r4, fp, ip, lr, pc}
40: e3a0401e mov r4, #30 ; 0x1e
44: e24cb004 sub fp, ip, #4 ; 0x4
48: e1a00004 mov r0, r4
4c: ebffffeb bl 0 <fib>
50: e1a00004 mov r0, r4
54: e91ba810 ldmdb fp, {r4, fp, sp, pc}
```

**Listing 1.4.** Complete Assembly Code

```
1: /* function to determine whether status bits should ne set */
2: bool SetStatusB(int i) { // i is the PC of instruction; function that tells whether
   status bits should be set
3: // comparisons for function fib
4: if (i==4) { // setting status bits for instruction cmp at 4 [0x4]
5:     return true ;
6: }
7: if (i==32) { // setting status bits for instruction cmp at 32 [0x20]
8:     return true ;
9: }
10: if (i==36) { // setting status bits for instruction cmp at 36 [0x24]
```

```

11:     return true ;
12: }
13: // comparisons for function main
14: return false ;
15: }
16:
17: /* comparisons for instructions used in the program */
18: bool cmpU(int i) {
19:     /* comparisons for function fib starting 0 ending 52 */
20:     if (i==4) return val[r2]==UNKNOWN||val[r0]==UNKNOWN; // [0x4]
21:     if (i==32) return val[r2]==UNKNOWN; // [0x20]
22:     if (i==36) return val[r2]==UNKNOWN||val[ip]==UNKNOWN; // [0x24]
23:     /* comparisons for function main starting 56 ending 84 */
24:     return false; // none if not found
25: } // end comp of instruction
26:
27: /* setcmp for instructions used in the program */
28: void setcmp(int i,bool n1,bool n2) {
29:     /* res_comp for function fib starting 0 ending 52 */
30:     if (i==4) { // instruction cmp r2, r0 at 4 [0x4]
31:         cmple=n1;
32:     }
33:     if (i==32) { // instruction cmp r2, #30 at 32 [0x20]
34:         cmple=n1;
35:     }
36:     if (i==36) { // instruction cmple r2, ip at 36 [0x24]
37:         cmple=n1;
38:     }
39:     /* res_comp for function main starting 56 ending 84 */
40: } // end setcmp of instruction
41:
42: bool NDcmp(int i) {
43:     return SetStatusB(i) && cmpU(i) ;
44: }
45:
46: /* setcmp for instructions used in the program */
47: void setcmp(int i,bool n1,bool n2) {
48:     /* setcmp for function fib starting 0 ending 52 */
49:     if (i==4) { // instruction cmp r2, r0 at 4 [0x4]
50:         cmple=n1;
51:     }
52:     if (i==32) { // instruction cmp r2, #30 at 32 [0x20]
53:         cmple=n1;
54:     }
55:     if (i==36) { // instruction cmple r2, ip at 36 [0x24]
56:         cmple=n1;
57:     }
58:
59:     /* res_comp for function main starting 56 ending 84 */
60:
61: } // end setcmp of instruction

```

**Listing 1.5.** C Code for SetStatusB and cmpU

```

1: void update() { // update function
2:     int nextpc,nextfp,tmp;
3:     /*
4:      *updates for function fib starting 0 ending 52
5:      */
6:     if (val[pc]==0) { // Instruction mov r2, #2 at 0x0
7:         nextpc=val[pc]+4;
8:         if (!is_abstracted(val[pc])) { // effect of instruction is null if abstracted
9:             val[r2]=(2);
10:        }
11:        SET(0,-1,1); // instruction scheduled is 0, no memory access and scheduled
12:    } // end mov at 0x0
13:    if (val[pc]==4) { // Instruction cmp r2, r0 at 0x4
14:        nextpc=val[pc]+4;
15:        if (!is_abstracted(val[pc])) { // effect of instruction is null if abstracted
16:            // Should set the Z and N and C bits
17:            if ((val[r2]-(val[r0]))<=0) cmple=1 ; else cmple=0;
18:        }
19:        SET(4,-1,1); // instruction scheduled is 4, no memory access and scheduled
20:    } // end cmp at 0x4
21:
22:    ...

```



```

23:
24: if (val[pc]==20) { // Instruction movgt pc, lr at 0x14
25:     nextpc=val[pc]+4;
26:     if (gt()) {
27:         if (!is_abstracted(val[pc])) { // effect of instruction is null if abstracted
28:             if (val[lr]==UNKNOWN) {
29:                 val[pc]=UNKNOWN;
30:             }
31:             else {
32:                 nextpc=(val[lr]);
33:             }
34:         }
35:         SET(20,-1,1); // instruction scheduled is 20, no memory access and scheduled
36:     }
37:     else SET(20,-1,0) ; // instruction not scheduled, no mem access
38: } // end movgt at 0x14
39: if (val[pc]==24) { // Instruction add r2, r2, #1 at 0x18
40:     nextpc=val[pc]+4;
41:     if (!is_abstracted(val[pc])) { // effect of instruction is null if abstracted
42:         if (val[r2]==UNKNOWN) {
43:             val[r2]=UNKNOWN;
44:         }
45:         else {
46:             val[r2]=(val[r2]+1);
47:         }
48:     }
49:     SET(24,-1,1); // instruction scheduled is 24, no memory access and scheduled
50: } // end add at 0x18
51:
52: ...
53:
54: if (val[pc]==32) { // Instruction cmp r2, #30 at 0x20
55:     nextpc=val[pc]+4;
56:     if (!is_abstracted(val[pc])) { // effect of instruction is null if abstracted
57:         // Should set the Z and N and C bits
58:         if ((val[r2]-(30))<=0) cmple=1 ; else cmple=0;
59:     }
60:     SET(32,-1,1); // instruction scheduled is 32, no memory access and scheduled
61: } // end cmp at 0x20
62: if (val[pc]==36) { // Instruction cmple r2, ip at 0x24
63:     nextpc=val[pc]+4;
64:     if (le()) {
65:         if (!is_abstracted(val[pc])) { // effect of instruction is null if abstracted
66:             // Should set the Z and N and C bits
67:             if ((val[r2]-(val[ip]))<=0) cmple=1 ; else cmple=0;
68:         }
69:         SET(36,-1,1); // instruction scheduled is 36, no memory access and scheduled
70:     }
71:     else SET(36,-1,0) ; // instruction not scheduled, no mem access
72: } // end cmple at 0x24
73:
74: ...
75:
76: if (val[pc]==48 && (!le())) { // Instruction ble 18, at 0x30
77:     nextpc=val[pc]+4;
78:     SET(48,-1,0) ; // instruction scheduled, no mem access, no branching
79: } // end ble at 0x30 [cond false]
80: if (val[pc]==48 && le()) { // Instruction ble 18, at 0x30
81:     nextpc=24; // to 0x18
82:     SET(48,-1,1) ; // instruction scheduled, no mem access, branching
83: } // end ble at 0x30 [cond true]
84: if (val[pc]==52) { // Instruction mov pc, lr at 0x34
85:     nextpc=val[pc]+4;
86:     if (!is_abstracted(val[pc])) { // effect of instruction is null if abstracted
87:         if (val[lr]==UNKNOWN) {
88:             val[pc]=UNKNOWN;
89:         }
90:         else {
91:             nextpc=(val[lr]);
92:         }
93:     }
94:     SET(52,-1,1); // instruction scheduled is 52, no memory access and scheduled
95: } // end mov at 0x34
96:
97: /*
98: end of updates for function fib
99: */
100:

```

```

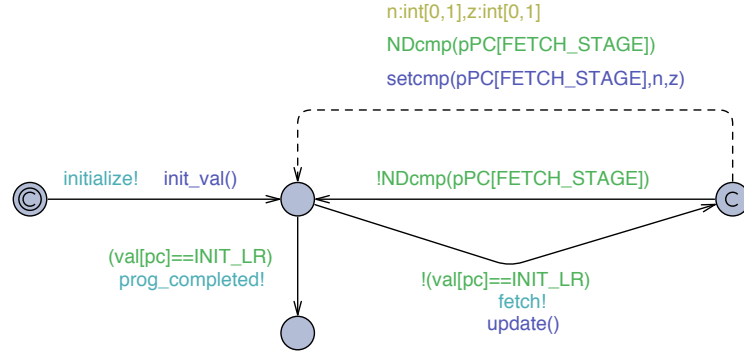
101:  /*
102:   * updates for function main starting 56 ending 84
103:   */
104:  if (val[pc]==56) { // Instruction mov ip, sp at 0x38
105:      nextpc=val[pc]+4;
106:      if (!is_abstracted(val[pc])) { // effect of instruction is null if abstracted
107:          if (val[sp]==UNKNOWN) {
108:              val[ip]=UNKNOWN;
109:          }
110:          else {
111:              val[ip]=(val[sp]);
112:          }
113:      }
114:      SET(56,-1,1); // instruction scheduled is 56, no memory access and scheduled
115:  } // end mov at 0x38
116:  if (val[pc]==60) { // Instruction stmb sp!,{r4,fp,ip,lr,pc,} at 0x3c
117:      nextpc=val[pc]+4;
118:      // push should first decrease val[pc] and then store in stack(val[pc])
119:      push(val[pc]);
120:      push(val[lr]);
121:      push(val[ip]);
122:      push(val[fp]);
123:      push(val[r4]);
124:      SET(60,-1,1); // instruction scheduled is 60, no memory access
125:  } // end stmb at 0x3c
126:
127:  ...
128:
129:  if (val[pc]==76) { // Instruction bl 0, (unconditional) at 0x4c
130:      nextpc=0; // to 0x0
131:      val[lr]=80;
132:      SET(76,-1,1); // instruction scheduled, no mem access, branching
133:  } // end bl at 0x4c
134:  if (val[pc]==80) { // Instruction mov r0, r4 at 0x50
135:      nextpc=val[pc]+4;
136:      if (!is_abstracted(val[pc])) { // effect of instruction is null if abstracted
137:          if (val[r4]==UNKNOWN) {
138:              val[r0]=UNKNOWN;
139:          }
140:          else {
141:              val[r0]=(val[r4]);
142:          }
143:      }
144:      SET(80,-1,1); // instruction scheduled is 80, no memory access and scheduled
145:  } // end mov at 0x50
146:  if (val[pc]==84) { // Instruction ldmb fp,{r4,fp,sp,pc,} at 0x54
147:      nextpc=val[pc]+4;
148:      nextpc=stack(val[fp]-4);
149:      val[sp]=stack(val[fp]-8);
150:      nextfp=stack(val[fp]-12);
151:      val[r4]=stack(val[fp]-16);
152:      val[fp]=nextfp;
153:      SET(84,-1,1); // instruction scheduled is 84, no memory access
154:  } // end ldmb at 0x54
155:
156:  /*
157:   * end of updates for function main
158:   */
159:
160:  val[pc]=nextpc;
161:  } // end update

```

**Listing 1.6.** C Program

The generic automaton to simulate a program  $p_{\perp}$  is given in Fig. 2. We assume that the main function of the program  $p_{\perp}$  is called by another program and a particular value `INIT_LR` gives the return point. The automaton *Prog* performs some initialization (`init_val()`) and then computes the next state until the end of the program is reached: this is when the value of the `pc` register is equal to the return point `INIT_LR` (guard `val[pc]=INIT_LR`). To simulate each instruction, the automaton *Prog* performs the following steps:

1. feed the current instruction  $\iota$  to the first stage of the pipeline when it is empty (to do so it has to synchronize with the first stage of the pipeline, on the **fetch!** channel) and compute the next state (**update()** function). This also sets the next value of register **pc**. The result of **update()** is that the number of the current instruction is stored into the variable **pPC[FETCH\_STAGE]** where **FETCH\_STAGE** is the number of the first stage of the pipeline (0);
2. if the instruction  $\iota$  in **pPC[FETCH\_STAGE]** is an undetermined comparison (**NDcmp(pPC[FETCH\_STAGE])** evaluates to **TRUE**), the upper dashed transition is taken: Player 2 chooses two values  $n$  and  $z$  and the predicates that must be set (**cmple**, **cmplt**, etc) are set by **setcmp** (Listing 1.5). If  $\iota$  does not set any flag or the outcome is determined by the current state (the operands are all known), the middle transtion is taken (Player 2 does not have to play).



**Fig. 2.** Generic Automaton *Prog* to Simulate a Program

## 4 Model of the Hardware

In this section we give a UPPAAL model for the architecture of the pipelined processor ARM9 and for the caches.

### 4.1 Model of the Pipeline

Each stage of the pipeline contains an instruction (and some other information). The information for each stage of the pipeline are stored in arrays: **pPC[k]** gives the number of the instruction in stage  $k$ ; **Todo[k]** is a boolean value and indicates whether the instruction **pPC[k]** is scheduled (some instructions are conditional and are skipped); **dataAdr[k]** contains the address<sup>11</sup> of the memory

<sup>11</sup> For multiple loads and stores, this should be a range of addresses; this information is used only for determining whether a stall should occur in the pipeline. For multiple

cell referenced by instruction `pPC[k]` (−1 if none). There are 5 stages in the pipeline of the ARM9:

- stage 1: this is the *fetch* stage. It fetches the next instruction (pointed to by the `pc` register) from the cache (or main memory) and this instruction becomes the current instruction of stage 1;
- stage 2: *decode* stage. Decodes the instruction in stage 2;
- stage 3: *execute* stage. Carries out the computation (addition, comparisons, etc) of the instruction in stage 3;
- stage 4: *memory* stage. Carries out the transfers (from registers to main memory or main memory to registers) of the instruction in stage 4;
- stage 5: *writeback* stage. Writes the value of registers that are (“writeback”) operands of the instruction in stage 5.

An instruction  $\iota$  enters the pipeline at stage 1. It is transferred from stage  $i$  to  $i + 1$  as soon as possible. When it exits stage 5, it is completed. The execution of a program is completed when its last instruction is completed.

**Pipeline Stalls.** The goal of *pipelining* is to split the execution of an instruction into different simple steps. The idea being that each step can be carried out concurrently for different instructions: while stage 1 fetches the next instruction  $\iota_k$ , stage 2 decodes instruction  $\iota_{k-1}$ , etc. It may happen that the simple steps of some sequences of instructions cannot be carried out concurrently. A *pipeline stall* is a situation when one stage  $i$  of the pipeline cannot perform its computation because it has to wait for another stage  $j > i$  to complete its computation. An example is when the execution of an instruction at stage 3 (execute) has an operand which is set in stage 4 (memory).

The sequence of instructions of lines 0 and 4 will result in a pipeline stall at stage 3 for instruction 4: when instruction 4 ( $r2 := r0 - r1$ ) is ready to execute at stage 3, it has to wait for instruction 0 to complete (at stage 4) because instruction 0 loads the value of memory cell `r1` into `r0`.

```
0:  ldr r1, [r0]
4:  sub r2, r0, r1
8:  ...
c:  ldm r13, {r1,r2,r3}
10: add r4, r3, #1
14: ...
```

**Listing 1.7.** Stalls

Thus instruction 4 stalls for one cycle<sup>12</sup> at stage 3. The situation for instructions `c` and `10` can even result in more than one cycle delay. The `ldm` instruction (line `c`) is a *multiple load* instruction. It loads the registers `r1`, `r2` and `r3` with the contents of memory cells pointed to by `r13`. Stage 4 performs the loads, but only one per cycle. Thus instruction `10` stalls for 3 cycles at stage 3.

A pipeline stall may occur depending on: (i) the type of the instruction at stage 3, and the type of the instruction at stages 4 and 5; (ii) the registers (and memory addresses) used by the instructions at the corresponding stages.

---

loads and stores, we force a stall in a pipeline until the end of the multiple load-s/stores instruction. This is a safe encoding as the ARM9 does not exhibit timing anomalies.

<sup>12</sup> We assume that the content of memory cell was in the cache and it takes one cycle to be fetched.

**Branch Prediction.** When a conditional branch instruction enters the pipeline, the next instruction to flow in is determined by the truth value of the condition. This value might not yet be available when the branch instruction is in the first stage of the pipeline. If the condition is determined by the value of a variable which is not in the cache, it might take a few cycles before the result becomes available. In this case, we should *stall* until the outcome of the comparison is computed. This might however be inefficient.

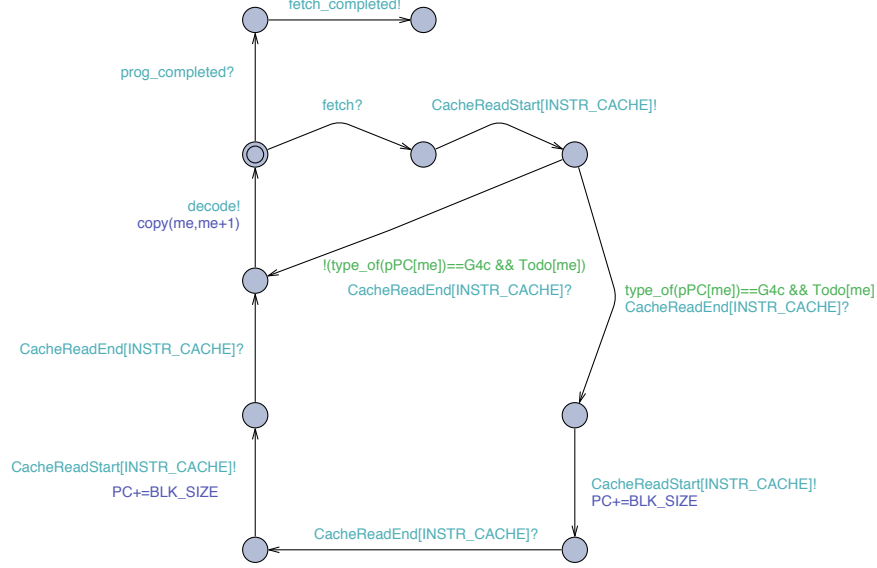
Some heuristics can be applied to guess the most plausible next instruction after a conditional branching. After the *prediction*, the chosen instruction flows in the pipeline. If the guess was right the result is shortest execution time for this part of the program. If the guess was wrong, the computations of the mistakenly taken branch have to be undone, and the pipeline flushed which results in a longer execution time. We do not discuss here the choice of a good heuristics, but there are a few options that give good results on *average*.

In our model we follow [20] and model the heuristics for branch prediction by: in a conditional branch, a branch is never taken (other heuristics can be accommodated for in our model).

**UPPAAL Pipeline Model.** The timed automata models we introduce are close to the ones proposed in [20]. However there are some differences as we do not have the same model for the program.

The timed automata for each stage (ARM9, 5 stages) are depicted on Fig. 3 and Fig. 4. The stage modelled by each automaton can be inferred by the synchronization channel from the initial state (e.g., `decode?`). The first stage of the pipeline is of particular importance as it models the case of a wrong guess in an branch prediction. The automaton of Fig. 3 models the following behaviour:

1. the automaton accepts a `fetch?` synchronization when it is idle;
2. after accepting an instruction (`fetch?` synchronizes with `fetch!` in the automaton *Prog* of Fig. 2), it actually fetches the instruction from main memory via the *instruction cache* (`CacheReadStart[INSTR.CACHE]!`, where `INSTR.CACHE` is the ID of the instruction cache);
3. when the instruction has been read from the cache or main memory, there are two options:
  - (a) the instruction  $\iota$  to be processed is a *conditional branch* (condition `type_of(pPC[me])==G4c`) and the variable `Todo[me]` indicates whether the condition was evaluated to TRUE or FALSE. In case it is a conditional branch and the condition was TRUE, we simulate two “instruction read from the cache” steps: indeed our branch prediction algorithm is “never branch” and thus if it happened that we had to branch, we should simulate a pipeline `flush`. As we do not execute the instructions in the pipeline (but rather when we feed the first stage of the pipeline), this can be modelled by reading the next two instructions (the “never branch” prediction) without executing them, and then resuming the simulation from the target address of the branch instruction.



**Fig. 3.** Timed Automata Model of the ARM9 Pipeline

- (b) the instruction to be processed is not a conditional branching or the condition was evaluated to FALSE; in this case the prediction was right and nothing has to be undone.

After an instruction has been fetched in the **fetch** stage, it is fed to the next stage of the pipeline. This is modelled by the **decode!** synchronization and the **copy(me,me+1)** transition. **copy(me,me+1)** copies the information in **pPC[me]**, **Todo[me]** and **dataAdr[me]** to the next stage **me+1**.

The memory stage automaton is a bit more involved than the others as it has to take into account different options: if the instruction is a memory transfer (**type\_of(pPC[me-1]) == G2LDR** or **type\_of(pPC[me-1]) == G2STR**) and is scheduled (**Todo[me-1]** is TRUE) a synchronization with the data cache is requested.

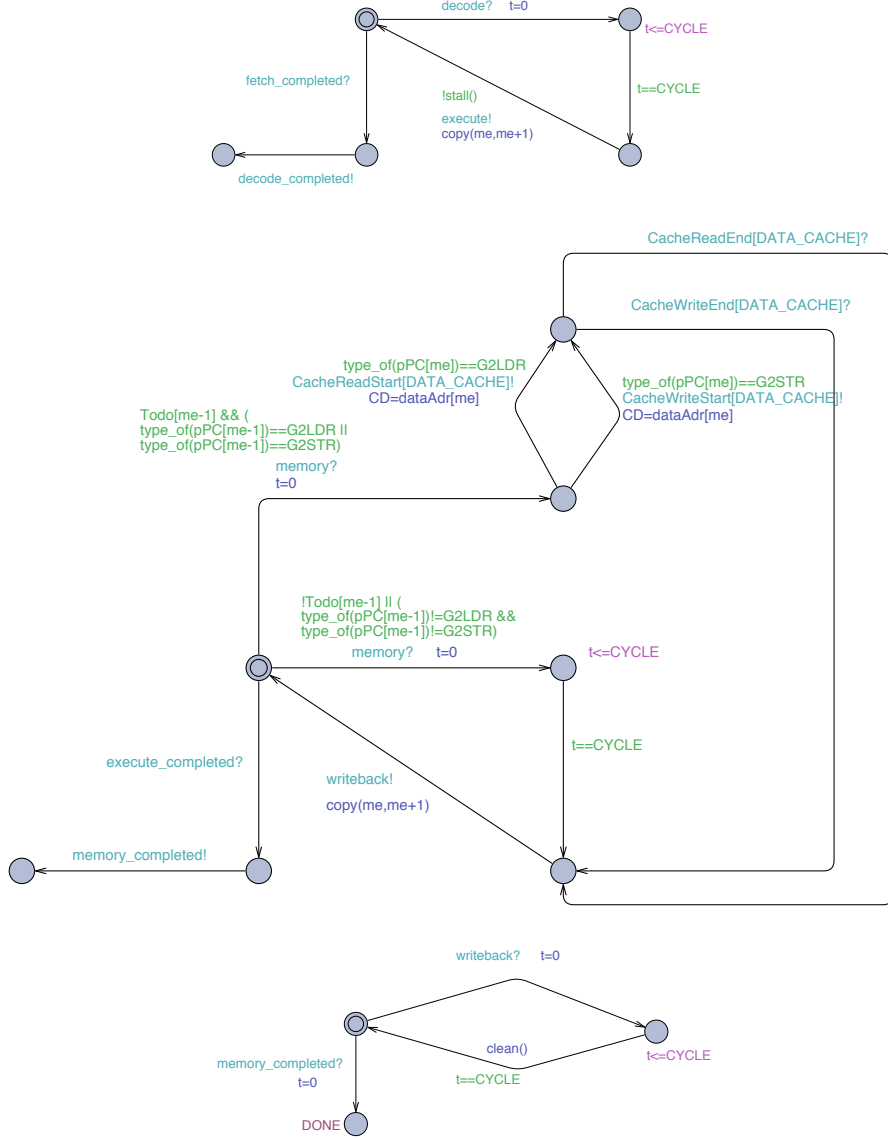
The type of the instructions is given by a UPPAAL function **type\_of**. The duration is also given by a function **dur()** (used in the execute stage).

## 4.2 Model of the Caches

A *cache* is a fast memory device. It is characterized by its size  $K$  (usually in Kbytes), the length of a cache *line* ( $B$  in Bytes) and the number of cache lines  $L = \frac{K}{B}$ .

The main memory  $\mathcal{M}$  of a computer is divided into blocks equal to the length of the cache line. We let  $\mathcal{M} = \{m_0, m_1, \dots, m_n\}$ .

The *associativity* of a cache determines where a memory block can reside.



**Fig. 4.** Timed Automata Model of the ARM9 Pipeline

- *fully associative*: a block can be in any line;
- *direct mapped*: a block can be in one line;
- *j-way*: a block can be in  $j$  different lines; in this case the cache is partitioned into  $\frac{L}{j}$  different sets. Fully and direct mapped are particular instances of  $j$ -way caches. The partition induced by the  $j$ -way cache is denoted  $\mathcal{P} = \{P_1, \dots, P_{\frac{L}{j}}\}$ .

The set of lines a memory can reside in is given by a mapping  $\kappa : \mathcal{M} \rightarrow \mathcal{P}$ .

The replacement policy determines which block to eject from memory when the cache is full. The most common policies are:

- LRU: least recently used;
- FIFO: first-in first-out;
- alternate and mixed and even random are permitted but not easily predictable.

Handling *writing* requests is also a distinctive feature of a cache.

- handling write *hits*:
  - **write trough**: write cache and memory
  - **write back**: write cache; need for a dirty bit which is taken care of when ejecting a line from the cache;
- handling write *misses*:
  - **write allocate**: write memory and fetch into cache;
  - **write no allocate**: write memory (no fetch).

In this paper we model a cache with FIFO replacement policy and assume write allocate on a write/miss.

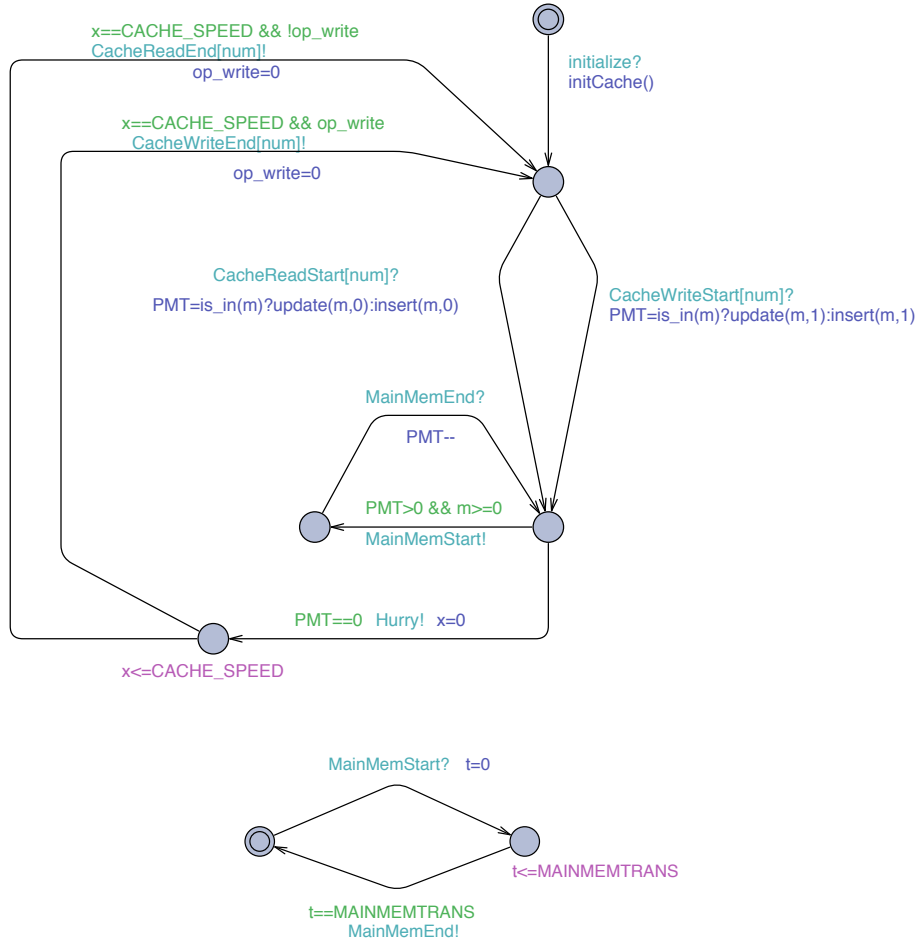
**UPPAAL Cache Model.** The automaton modeling the behaviour of the cache (together with the model of the main memory automaton) is given in Fig. 5.

After performing some initializations (`initCache()`, setting the initial state of the cache), it accepts either write or read requests. Depending on the request, and whether a cache line is dirty or not, a number of memory transactions (**PMT**) are needed to fetch the content of memory cell `m`. Each such transaction is performed one after the other. When it is completed the transfer from the cache to the register of the processor takes place and require **CACHE\_SPEED** time units.

## 5 Tool Chain and Case Studies

We have applied the previous framework to a number of benchmarks from Mälardalen University.





**Fig. 5.** Timed Automata Model for the Caches

**Tool Chain.** The tool chain to compute WCET is depicted on Fig. 6. The component we have developed are ARM2UPP and PATCH\_UPP:

- ARM2UPP takes as input a program in assembly (`file.arm`) that has been annotated with the comparisons operators for each instruction that sets a status bit. It generates four files:
  - `file.{xml,q}` that contain respectively the UPPAAL network automata (and functions like `update()` etc) modeling the execution of the program on the architecture of the ARM9 and the UPPAAL queries to compute/check the WCET;
  - `file-reach` is an executable obtained by compiling `file-reach.cpp`; this latter file is a C++ program that simulates the program in `file.arm`. `file-reach` always terminates. However, early termination can be forced by passing some parameters (maximal number of states, maximal number of split cases). In case the number of split cases is too large (e.g.,  $2^{50}$  for Bubble Sort), it is possible to add some information in the file `file-reach.cpp` like constraints on the outcome of an unknown comparison. This step may be iterated several times. When it is completed the file `file.info` contains some useful information (like maximal stack size, etc).
  - `file-equiv` is an executable obtained by compiling `file-equiv.cpp`; this program checks whether an abstraction mapping (which is given by a function) is valid or not (implements the algorithm of section 2.).
- PATCH\_UPP modifies some constants in `file.xml` to incorporate the information from `file.info` (like stack size) and can also include the function of abstracted instructions (if it has been declared valid).

**UPPAAL-TiGA Queries.** In order to compute the WCET of a program, we can check whether the program always terminates within  $k$  time units. This can be computed using a binary search with UPPAAL. The drawback of this check is that some deadlock may occur in the system, yielding a biased value of the WCET.

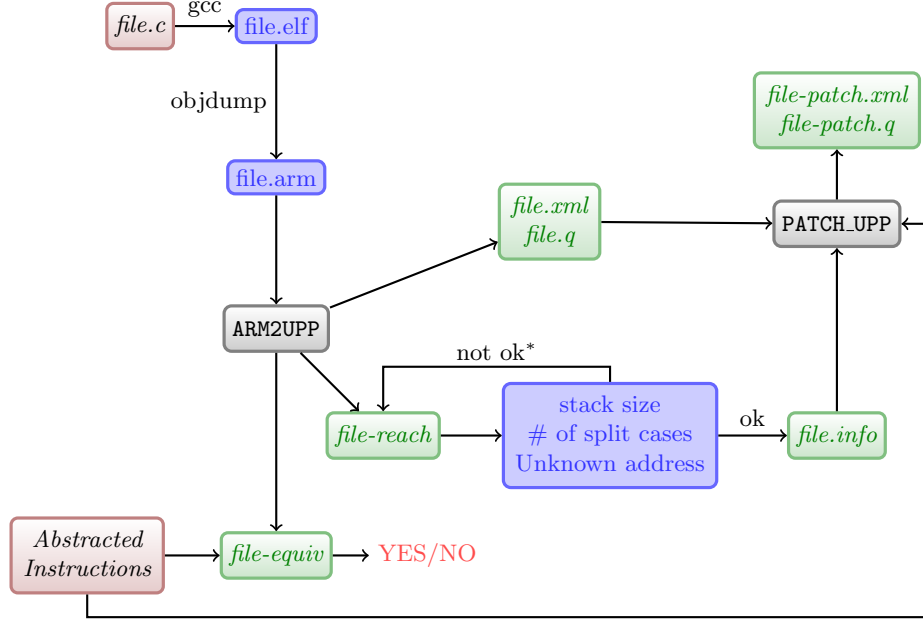
An alternative way of computing the WCET is check a *control* property: “Can Player 1 enforce termination of the program and if yes, what is the best duration he can guarantee?” This optimal time reachability control objective can be checked in one query (see [22]) with UPPAAL-TiGA, provided we know an upper of the WCET. This can be roughly over-estimated on the program (we have not implemented this part yet). Optimal reachability of a location 1 is then specified by the control objective:

`control(#n,0) : A [ true U 1 ]`

if `#n` is a rough upper bound of the WCET<sup>13</sup>.

Program termination in the UPPAAL model happens when the location `DONE` is reached in the `writeBackStage` automaton (last stage of the pipeline). Thus the control property we check is:

<sup>13</sup> If `#n` is not large enough, UPPAAL-TiGA result will be “not controllable”.



**Fig. 6.** Tool Chain Overview

```
control(#n,0) : A [ true U WriteBackStage.DONE ]
```

**case Studies & Results** We have applied the framework described in Fig. 6 to a number of benchmark programs from Mälardalen University. We could not analyse the full set of programs because of the current limitations of our tools:

- floating point operations are not supported yet;
- a few operators (e.g., `ror`) of the ARM9 assembly language are not supported yet.

There are not many published results about the actual WCET of the benchmarks (or when there are, the hardware parameters, cache speed, etc are not given). To evaluate the relevance of our method, we compare our results to the ones obtained with the METAMOC method [20].

There are 15 programs that can be analysed by METAMOC using a concrete instruction cache and an “always miss” data cache. Only 7 of the 15 programs can be analysed with both a concrete instruction and data cache. Using our encoding and tool chain, we could analyse 13 out of these 15 programs (two of them contains unsupported operations) with concrete caches. Moreover, the time/space needed to compute the results is very small compared to the resources used in METAMOC (32GB RAM computer). Table 1 give the values of WCET for each program, and the time for UPPAAL-TiGA to compute the result. The time needed to compute the intermediary files is negligible. The timing specification

of the caches are: `CACHE_SPEED=1` (processor) cycle is the same as the processor speed, and a memory transaction takes 10 processor cycles. The UPPAAL files are available from <http://www.irccyn.fr/franck/wcet>.

**Energy/Power Consumption Optimization.** The last column of Table 1 gives the percentage of time the processor can run at a slower clock rate (1/4th of its fastest speed) without any impact on the WCET: this is due to the initial transient phase of the execution of a program where instructions are loaded into the cache. For some small programs the result is impressive (22% for `janne-complex`). To do this we just add a automaton to the network that switches the rate from 4 to 1 after a certain amount of time. Another interesting and easy computation that can be done, is to fix the time the processor runs at a slower rate (in the initial phase) and compute the optimal time to reach the end the program (which is the WCET) under this constraint.

| Program                        | loc <sup>†</sup> | N <sup>‡</sup> | UPPAAL-TiGA<br>time/space | WCET   | Abs <sup>*</sup> | Low<br>Power   |
|--------------------------------|------------------|----------------|---------------------------|--------|------------------|----------------|
| <b>Single-Path Programs</b>    |                  |                |                           |        |                  |                |
| fac                            | 26               | 0              | 0.35s/6.91MB              | 1883   | 4/34             | 26/1.3%        |
| fib                            | 74               | 0              | 0.25s/5.68MB              | 571    | 4/22             | 26/4.5%        |
| janne-complex*                 | 65               | 0              | 0.54s/7.76MB              | 792    | 0/23             | <b>176/22%</b> |
| matmult*                       | 162              | 0              | 119.2s/936.75MB           | 614827 | <b>31/107</b>    | 800/0.001%     |
| jfdct                          | 374              | 0              | 7.13s/55.99MB             | 49017  | <b>394/454</b>   | 108/0.22%      |
| expint(50,1)                   | 81               | 0              | 6.08s/59.16MB             | 65042  | 0/124            | 70/1.7%        |
| expint(50,21)                  | 81               | 0              | 3.65s/43.21MB             | 41015  | 0/124            | 71/1.7%        |
| fdct                           | 238              | 0              | 2.83s/26.79MB             | 26099  | 0/286            | 90/0.3%        |
| edn*                           | 284              | 0              | 22.28s/230.98MB           | 62968  | 0/460            | 26/0.04%       |
| recursion*                     | 41               | 0              | 2.68s/28.82MB             | 10335  | 0/38             | 32/0.3%        |
| <b>Multiple-Paths Programs</b> |                  |                |                           |        |                  |                |
| bs                             | 174              | 5              | 0.52s/6.52MB              | 366    | 0/22             | <b>30/8.2%</b> |
| cnt*                           | 115              | 100            | 100.25s/377.02MB          | 6483   | 0/82             | 40/0.06%       |
| insertsort*                    | 91               | 675            | 9.36s/81.27MB             | 27061  | 0/53             | 400/1.4%       |
| ns*                            | 497              | 625            | 12.38s/110.92MB           | 43239  | 0/41             | 32/0.0007%     |

<sup>†</sup>lines of code in the C source file    <sup>‡</sup>N = Max number of Player 2 moves along a path

\*Abstracted Instr./Instr.                      \*Program selected for the WCET Challenge 2006 [24]

**Table 1.** Results (C programs compiled with `gcc -O2`)

## 6 Conclusion

In this paper we have presented a framework based on timed games and the model checker UPPAAL-TiGA to compute WCET for programs running on architectures featuring pipelines and caches.

The results we have obtained support the claim that model checking is adequate for computing WCET. Moreover UPPAAL-TiGA could be tuned to handle WCET computation more efficiently: *priorities* between processes can reduce unnecessary interleavings and there are not yet implemented in UPPAAL-TiGA (though they are in UPPAAL); a lot of time is spent *checking* whether a new state has already been encountered: this will never be the case in the programs we check (otherwise they would be an infinite loop). Disabling this check would also reduce the time to compute the results. Of course, a program like Bubble Sort remains beyond the scope of analysis within our framework. Nevertheless, what we advocate is the combination of different techniques to solve the WCET problem: *abstract interpretation* (AI) combined with *Integer Linear Programming* (ILP) have given very good results [11] but this method is yet to prove that: (1) it can be *easily* adapted to different processors and (2) it can take into account *power* related features (like change of speed of the processor).

Our ongoing work focuses on two aspects:

1. extend the set of instructions supported by our compiler and provide models for other architectures (like ARM11);
2. add a *pre-processing* step to prune the execution tree of the program. The goal of this step is to reduce the number of paths of the program still preserving the paths giving the WCET. This step can be carried out using ILP techniques, or *counter-example guided abstraction refinement* (CEGAR) methods [25].

#### **Acknowledgements.**

The author would like to thank Bernard Blackham (NICTA, Sydney) and Gernot Heiser (NICTA, Sydney) for their helpful comments and support.

## **References**

1. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D.B., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P.P., Staschulat, J., Stenström, P.: The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embedded Comput. Syst.* **7**(3) (2008)
2. Rapita Systems Ltd.: Rapita Systems for timing analysis of real-time embedded systems. <http://www.rapitasystems.com/>.
3. Bernat, G., Colin, A., Petters, S.M.: pWCET a Toolset for automatic Worst-Case Execution Time Analysis of Real-Time Embedded Programs. In: *Proceedings of the 3rd Int. Workshop on WCET Analysis, Satellite Workshop of the Euromicro Conference on Real-Time Systems*, Porto, Portugal (2003)
4. Rieder, B., Puschner, P., Wenzel, I.: Using Model Checking to Derive Loop Bounds of General Loops within ANSI-C Applications for Measurement Based WCET Analysis. In: *Proceedings of the 6th Int. Workshop on Intelligent Solutions in Embedded Systems (WISES'08)*, Regensburg, Germany (2008)
5. Tidorum Ltd.: Bound-T time and stack analyser. <http://www.bound.com/>.
6. OTAWA: Open Tool for Adaptive WCET Analyses. <http://www.otawa.fr/>.

7. Prantl, A., Schordan, M., Knoop, J.: TuBound - A Conceptually New Tool for WCET Analysis. In: Proceedings of the 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis (WCET'08), Prague, Czech Republic (July 2008)
8. Li, X., Liang, Y., Mitra, T., Roychoudhury, A.: Chronos: A Timing Analyzer for Embedded Software. *Science of Computer Programming* **69**(1–3) (2007) Special Issue on Experimental Software and Toolkit.
9. Engblom, J., Ermedahl, A., Nolin, M., Gustafsson, J., Hansson, H.: Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer* **4**(4) (October 2003) 437–455
10. AbsInt Angewandte Informatik: aiT Worst-Case Execution Time Analyzers. <http://www.absint.com/ait/>.
11. Ferdinand, C., Heckmann, R., Wilhelm, R.: Analyzing the worst-case execution time by abstract interpretation of executable code. In Broy, M., Krüger, I.H., Meisinger, M., eds.: ASWSD. Volume 4147 of *Lecture Notes in Computer Science.*, Springer (2004) 1–14
12. Holsti, N., Gustafsson, J., Bernat, G., Ballabriga, C., Bonenfant, A., Bourgade, R., Cassé, H., Cordes, D., Kadlec, A., Kirner, R., Knoop, J., Lokuciejewski, P., Merriam, N., Michiel, M.D., Prantl, A., Rieder, B., Rochange, C., Sainrat, P., Schordan, M.: Wcet 2008 - report from the tool challenge 2008. In: Proceedings of the 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis (WCET'08), Prague, Czech Republic (July 2008)
13. Alur, R., Dill, D.: A theory of timed automata. *Theoretical Computer Science* **126**(2) (1994) 183–235
14. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. *Journal of Software Tools for Technology Transfer (STTT)* **1**(1-2) (1997) 134–152
15. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: QEST, IEEE Computer Society (2006) 125–126
16. Wilhelm, R.: Why ai + ilp is good for wcet, but mc is not, nor ilp alone. In Steffen, B., Levi, G., eds.: VMCAI. Volume 2937 of *Lecture Notes in Computer Science.*, Springer (2004) 309–322
17. Metzner, A.: Why model checking can improve wcet analysis. In Alur, R., Peled, D., eds.: CAV. Volume 3114 of *Lecture Notes in Computer Science.*, Springer (2004) 334–347
18. Hubert, B., Schoeberl, M.: Comparison of Implicit Path Enumeration and Model Checking Based WCET Analysis. In: Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis (WCET'09), Dublin, Ireland (July 2009)
19. Ouimet, M., Lundqvist, K.: The TASM Toolset: Specification, Simulation and Formal Verification of Real-Time Systems. [26] 126–130 (Tool Paper).
20. Dalsgaard, A.E., Olesen, M.C., Toft, M.: Modular execution time analysis using model checking. Master's thesis, Department of Computer Science, Aalborg University, Denmark (2009)
21. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: Uppaal-tiga: Time for playing games! [26] 121–125 (Tool Paper).
22. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Proc. of the 16th Int. Conf. on Concurrency Theory (CONCUR'05). Volume 3653 of *Lecture Notes in Computer Science.*, Springer (August 2005) 66–80
23. ARM Ltd.: ARM9 – ARM9 Processor Family Available from <http://www.arm.com/products/CPUs/families/ARM9Family.html>.

24. Mälardalen WCET Research Group: WCET Project – Benchmarks. <http://www.mrtc.mdh.se/projects/wcet/benchmarks.html>.
25. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* **50**(5) (2003) 752–794
26. Damm, W., Hermanns, H., eds.: Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings. In Damm, W., Hermanns, H., eds.: CAV. Volume 4590 of Lecture Notes in Computer Science., Springer (2007)